Objektorientierte Programmierung

Kapitel 12: Interfaces

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2012/13

http://www.informatik.uni-halle.de/~brass/oop12/

Inhalt

Einführung, Motivation (1)

- Java verwendet “Single Inheritance”, d.h. jede Klasse kann maximal eine Oberklasse haben.

  Genauer hat jede Klasse außer `Object` genau eine Oberklasse.

- Manche anderen Sprachen, wie z.B. C++, lassen “Multiple Inheritance” zu. Dort kann eine Klasse gleichzeitig von mehreren Oberklassen erben.

- Das kann zu unübersichtlichen Situationen führen, wenn z.B. ein Attribut auf verschiedenen Wegen an eine Klasse vererbt wird.

- Mit Interfaces bekommt Java eine Form von Multiple Inheritance, die einfacher zu überblicken ist.

Einführung, Motivation (2)

- Ein Interface ist eine abstrakte Klasse mit nur abstrakten Methoden.

  Formal ist ein Interface keine Klasse. Die Referenz-Typen in Java sind Klassen, Interfaces, und Arrays.

- D.h. es ist angegeben, welche Methoden mit welchen Argument- und Resultat-Typen es geben muss, aber es ist keine Implementierung für diese Methoden angegeben, d.h. kein Methoden-Rumpf.

- Eine Klasse kann
  - nur eine Oberklasse haben, aber
  - beliebig viele Interfaces implementieren.

    D.h. sie muss die im Interface angegebenen Methoden zur Verfügung stellen (so als wenn das Interface eine abstrakte Oberklasse wäre).

Einführung, Motivation (3)

- Interfaces sind Typen. Man kann also z.B. eine Variable von einem Interface-Typ deklarieren.
- Man kann keine direkten Instanzen von Interfaces anlegen.

  Sie sind ja abstrakt, d.h. den Methoden würde die Implementierung fehlen.
- Eine Klasse `C`, die ein Interface `I` implementiert, ist ein Untertyp von `I`.

  Da es sich nicht um Klassen handelt, spricht man von "Untertyp" und "Obertyp", aber diese verhalten sich z.B. bei Zuweisungen wie "Unterklasse" und "Oberklasse".
- Man kann also ein Objekt der Klasse `C` einer Variablen vom Typ `I` zuweisen.

Einführung, Motivation (4)

- Für viele Datenstrukturen gibt es
  - ein abstraktes Interface (das die Schnittstelle beschreibt) und
  - mehrere Klassen, die das Interface auf ganz unterschiedliche Arten implementieren.
- Z.B. gibt es für Listen von Objekten vom Typ `T` das Interface `List<T>`.

  Hier ist `T` ein Typ-Parameter, s. Kap. 14. Z.B. wäre `List<Person>` eine Liste von `Person`-Objekten. `List<T>` ist im Paket `java.util` definiert.
- Es gibt eine ganze Reihe von Klassen, die dieses Interface implementieren, z.B.
  - `ArrayList<T>`: speichert Listenelemente in Array.
  - `LinkedList<T>`: verkettet Listenknoten (`next`-Zeiger).

Einführung, Motivation (5)

- Da es für die Verwendung der Liste nur auf die zur Verfügung gestellten Methoden ankommt, deklariert man Variablen vom Typ `List<T>`.
- Wenn man eine konkrete Liste anlegt, muss man sich natürlich für eine Implementierung entscheiden:

```
List<Person> l = new LinkedList<Person>();
```

  Damit das funktioniert, muss man die Klassen aus dem entsprechenden Paket importieren, z.B. mit "`import java.util.*;`" (oben in der Java-Datei).

- Jede Implementierung hat ihre Stärken und Schwächen.

  Geschwindigkeit unterschiedlicher Operationen, Speicherplatz-Bedarf.

- Wenn man eine andere Implementierung möchte, braucht man nur die rechte Seite dieser Zuweisung zu ändern, der Rest des Programms bleibt unverändert.

Einführung, Motivation (6)

- Der `instanceof`-Operator kann rechts einen beliebigen Referenz-Typ haben, also auch ein Interface.
- Daher macht sogar ein Interface ganz ohne Methoden Sinn: Es kann benutzt werden, um Klassen zu markieren.
- Ein Beispiel ist das Interface `cloneable`:
  - Die von `Object` ererbte Methode `clone()` fragt ab, ob die Klasse des aktuellen Objektes das Interface `cloneable` implementiert.
  - Falls ja, wird das Objekt kopiert.
  - Falls nein, gibt es eine `CloneNotSupportedException`.

Inhalt

Syntax, Beispiel (1)

- Ein Interface kann Folgendes enthalten:
  - Abstrakte Methoden
  - Konstanten
  - Geschachtelte Klassen und Interfaces

    Geschachtelte Klassen und Interfaces können in dieser Vorlesung leider nicht mehr behandelt werden.
- Ein Interface kann nicht enthalten:
  - Statische Methoden (die können nicht `abstract` sein).
  - Attribute (das wäre schon Implementierung).
  - Konstruktoren (man kann keine Objekte erzeugen).

Syntax, Beispiel (2)

```
(1)  interface Abbrennplan {
(2)
(3)      // Zeitangabe bzgl. Anfang des Feuerwerks:
(4)      int ABSOLUTE_ZEIT = 1;
(5)      // Vom Anfang des vorigen Artikels:
(6)      int RELATIV_ANFANG = 2;
(7)      // Von Ende Brenndauer voriger Artikel:
(8)      int RELATIV_ENDE = 3;
(9)
(10)     // Weiteren Kanal anlegen (zeit in 0.1s):
(11)     int naechsterKanal(int zeitArt, int zeit);
(12)
(13)     // Artikelmenge fuer aktuellen Kanal erw.:
(14)     void artikelHinzu(Artikel art, int stueck);
(15)
(16)     ... // Abfrage-Methoden
(17) }
```

Syntax, Beispiel (3)

- Weil es um die Schnittstelle geht, sind alle Bestandteile eines Interfaces implizit `public`.

  Man könnte den Modifier `public` schreiben, es ist aber üblich, dies nicht zu tun.

- Das Interface selbst braucht nicht `public` zu sein, so kann man doch eine Einschränkung auf das Paket bekommen.

- Man läßt auch den Modifier "`abstract`" bei Methoden weg.

  Man dürfte ihn hinschreiben, aber er versteht sich bei einem Interface von selbst.

- Und entsprechend "`static final`" bei Konstanten.

  Sie sehen dann wie initialisierte Attribute aus. Ein Interface kann aber keine Attribute haben, nur Konstanten.

Syntax, Beispiel (4)

- Wenn man eine Klasse `C` mit Oberklasse `O` definieren will, die die Interfaces `I1`, `I2`, `I3` implementiert, schreibt man:

```
class C extends O implements I1, I2, I3 { ... }
```

- Die Teile für Oberklasse und implementierte Interfaces kann man einzeln weglassen, z.B.

```
class PlanArray implements Abbrennplan { ... }
```

- Falls die Klasse nicht `abstract` ist, muss sie alle Methoden aus allen implementierten Interfaces überschreiben und damit eine Implementierung (Methoden-Rumpf) angeben.

  Ist die Klasse mit dem Schlüsselwort `abstract` gekennzeichnet, erbt sie die Methoden aus dem Interface auch, braucht sie aber nicht selbst zu überschreiben. Dies muss dann aber in einer Subklasse geschehen, und nur von der (nicht-abstrakten) Subklasse können Objekte angelegt werden.

Syntax, Beispiel (5)

- Die implementierten Methoden müssen `public` sein.

  Weil sie im Interface implizit `public` sind, und die Zugriffsrechte in einer Subklasse nicht abgeschwächt werden dürfen (sonst wäre das Substitutionsprinzip verletzt, bzw. der Compiler könnte die Zugriffsrechte nicht überwachen).

- Falls man von verschiedenen Interfaces gleich benannte Methoden mit gleichen Parameter-Typen erbt,
  - gibt es keine Lösung, wenn die Resultat-Typen unterschiedlich sind (oder die Methoden unterschiedliche Dinge tun sollen),

    Wenn es einen gemeinsamen Subtyp der Resultat-Typen gibt, wäre es noch möglich.

  - ist es ansonsten in Ordnung, wenn eine Methode der Klasse mehrere ererbte Methoden implementiert.